Security in Active Networks

D. Scott Alexander[1], William A. Arbaugh[2], Angelos D. Keromytis[2], and Jonathan M. Smith[2]

[1] Bell Labs, Lucent Technologies
600 Mountain Avenue
Murray Hill, NH 07974 USA
salex@research.bell-labs.com

[2] Distributed Systems Lab
CIS Department, University of Pennsylvania
200 S. 33rd Str., Philadelphia, PA 19104 USA
{waa,angelos,jms}@dsl.cis.upenn.edu

Abstract. The desire for flexible networking services has given rise to the concept of "active networks." Active networks provide a general framework for designing and implementing network-embedded services, typically by means of a programmable network infrastructure. A programmable network infrastructure creates significant new challenges for securing the network infrastructure.
This paper begins with an overview of active networking. It then moves to security issues, beginning with a threat model for active networking, moving through an enumeration of the challenges for system designers, and ending with a survey of approaches for meeting those challenges. The Secure Active Networking Environment (SANE) realizes many of these approaches; an implementation exists and provides acceptable performance for even the most aggressive active networking proposals such as active packets (sometimes called "capsules").
We close the paper with a discussion of open problems and an attempt to prioritize them.

1 What is Active Networking ?

In networking architectures a design choice can be made between:

1. Restricting the actions of the network infrastructure to transport, and
2. easing those restrictions to permit on-the-fly customization of the network infrastructure.

The data-transport model, which has been successfully applied in the IP Internet and other networks, is called passive networking since the infrastructure (*e.g.,* IP routers) is mostly indifferent to the packets passing through, and their actions (forwarding and routing) cannot be directly influenced by users. This is not to say that the switches do not perform complex computations as a result of receiving or forwarding a packet. Rather, the nature of these computations cannot

dynamically change beyond the fairly basic configuration options provided by the manufacturer of the switch.

In contrast, *active* networking allows network-embedded functionality other than transport. For current systems, this functionality ranges from WWW proxy caches, multicasting [Dee89] and RSVP [BZB+97] to firewalls. Since each of these independently designed and supported functions could be carried out as an application of a more general infrastructure, the architecture of such *active* infrastructures is now being investigated aggressively.

The basic principle employed is the use of *programmability*, as this allows many applications to be created, including those not foreseen by the designers of the switch. There are a number of forms this programmability can take, including treating each packet as a program (active packets or "capsules") and programming or reprogramming network elements on-the-fly with select packets. Note that the latter approach subsumes the former, as a program may be loaded that treats all subsequent packets as programs.

1.1 Why is Active Network Security Interesting?

From a security perspective, a large scale infrastructure with user access to programming capabilities, even if restricted, creates a wide variety of difficult challenges. Most directly, since the basis of security is controlled access to resources, the increased complexity of the managed resources makes securing them much more difficult. Since "security" is best thought of as a mapping between a policy and a set of predicates maintained through actions, the policy must be more complex than, in as much as they exist, equivalent policies of present-day networks, resulting in an explosion in the set of predicates.

For example, the ability to load a new queuing discipline may be attractive from a resource control perspective, but if the queuing discipline can replace that of an existing user, the replacement policy must be specified, and its implementation carefully controlled through one or more policy enforcement mechanisms.

Additionally, such a scenario forces the definition of *principals* and objects with which policies are associated. When compared with the policy at a basic IP router (no principals, datagram delivery guarantees, FIFO queuing, *etc.*) it can be seen why securing active networks is difficult.

1.2 Virtual and Real Resources

As the role of active networking elements is to store, compute and forward, the managed resources are those required to store packets, operate on them, and forward them to other elements. The resources provided to various principals at any instant cannot exceed the real resources (*e.g.,* output port bandwidth) available at that instant. This emphasis on real resources and time implies that a conventional <*object, principal, access*> 3-tuple for an access control list (ACL) is inadequate.

To provide controlled access to real resources, with real time constraints, a fourth element to represent duration (either absolute or periodic) must be added,

giving <*object, principal, access, QoS guarantees*>. This remains an ACL, but is not "virtualized" by leaving time unspecified and making "eventual" access acceptable. We should point out that this new element in the ACL can be encoded as part of the *access* field. Similarly, we need not use an actual ACL, but we may use mechanisms that can be expressed in terms of ACLS and are better-suited for distributed systems.

2 Terminology

The term *trust* is used heavily in computer security. Unfortunately, the term has several definitions depending on who uses it and how the term is used. In fact, the U.S. Department of Defense's *Orange Book* [DOD85], which defined several levels of security a computer host could provide, defines *trust* ambiguously. The definition of *trust* used herein is a slight modification of that by Neumann [Neu95]. An object is defined as *trusted* when the object operates as expected according to design and policy. A stronger trust statement is when an object is *trustworthy*. A *trustworthy* object is one that has been shown in some convincing manner, *e.g.,* a formal code-review or formal mathematical analysis, to operate as expected. A *security-critical* object is one which the security — defined by a policy — of the system depends on the proper operation of the object. A security-critical object can be considered trusted, which is usually the case in most secure systems, but unfortunately this leads to an unnecessary profusion of such objects.

We note the distinction between *trust* and *integrity*: Trust is determined through the verification of components and the dependencies among them. Integrity demonstrates that components have not been modified. Thus integrity checking in a trustworthy system is about preserving an established trust or trust relationship.

2.1 Threat Model

An active network infrastructure is very different from the current Internet [AAKS98a]. In the latter, the only resources consumed by a packet at a router are:

1. the memory needed to temporarily store it, and
2. the CPU cycles necessary to find the correct route.

Even if IP [Pos81] option processing is needed, the CPU overhead is still quite small compared to the cost of executing an active packet. In such an environment, strict resource control in the intermediate routers was considered non-critical. Thus, security policies [Atk95] are enforced end-to-end. While this approach has worked well in the past, there are several problems. First, denial-of -service attacks are relatively easy to mount, due to this simple resource model. Attacks to the infrastructure itself are possible, and result in major network connectivity loss. Finally, it is very difficult to provide enforceable quality-of-service guarantees. [BZB+97]

Active Networks, being more flexible, considerably expand the threat possibilities, because of the increased numbers of potential points of vulnerability. For example, when a packet containing code to execute arrives, the system typically must:

- Identify the sending network element.
- Identify the sending user.
- Grant access to appropriate resources based on these identifications.
- Allow execution based on the authorizations and security policy.

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. Security violations occur when a policy is violated, *e.g.,* reading a private packet, or exceeding some specified resource usage. In the present-day Internet, intermediate network elements (*e.g.,* routers) very rarely have to perform any of these checks. This is a result of the best-effort resource allocation policies inherent in IP networking.

Denial-of-Service Attacks. Cryptographic mechanisms have proven remarkably successful for functions such as identification and authentication. These functions typically (although not necessarily) are used in protocols with a *virtual time* model, which is concerned with sequencing of events rather than more constrained sequencing of events with time limits (the *real time* model). The cases where time limits are observed are almost always for reasons of robustness, *e.g.,* to force eventual termination. Since such timeouts are intended for extreme circumstances, they are long enough so that they can cope with any reasonable delay.

In an environment where a considerable fraction (and perhaps eventually a majority) of the traffic will be continuous media traffic, security must include resource management and protection with an eye to preserving timing properties. In particular, a pernicious form of "attack" is the so-called "denial-of-service" attack. The basic principle applied in such an attack is that while wresting control of the service is desirable, the goal can be achieved if the opponent cannot use the service. This principle has been used in military communications strategies, *e.g.,* the use of radio "jamming" to frustrate an opponent's communications, and most recently in denying service to Internet Service Provider servers using a TCP *SYN* flood attack [Pan96, DRI96]. Another very effective (even crippling) attack on a computer system can occur due to scheduling algorithms which implicitly embed design assumptions.

To look at an example in some detail, consider the so-called "recursive shell" shown in Figure 1.

The shell script invokes itself. This is in fact a natural programming style, except that the process of invoking a shell script consists mainly of executing two heavyweight system calls, `fork()` and `exec()`, which, respectively, create a new copy of the current process and replace the current process with a new process created from an executable file. Since the program spends the majority of its time executing system calls, which in UNIX cause the operating system

to execute on behalf of the user (at high priority) the system's resources are typically consumed by this program (including CPU time and table space used for holding process control blocks).

With an active network element, it is easy to imagine situations where user programs (or errant system programs) run amok, and make the network elements useless for basic tasks. The solution, we believe, is to constrain *real* resources associated with active network programs. For example, if we limited the principal (*e.g.,* a "user") invoking the recursive shell script to 10% of the CPU time, or 10% of the system memory, the process would either limit its effects on the CPU to a 10% degradation, or fail to operate (since it could not invoke a new process) when it hit the table space limitation. Fortunately, a number of new operating systems [MMO+94, LMB+96] have appeared which provide the services necessary to contain one or more executing threads within a single scheduling domain.

```
#!/bin/sh
$0 #invoke ourselves
```

Fig. 1. A recursive shell script for UNIX

2.2 Challenges for the System Designer

Independent of the specific network architecture, the designer of a network has a set of tradeoffs they must make which define a "design space." We consider five here:

1. *Flexibility.* Flexibility is a measure of the system to perform a variety of tasks.
2. *Usability.* Usability is a measure of the ease with which the system can be used for its intended task(s).
3. *Performance.* The system will have some quantitative measures by which it is evaluated, such as throughput, delay, delay variation.
4. *Cost.* A networking system will have quantifiable economic costs, such as costs for construction, operation, maintenance and continuing improvements.
5. *Security.* Since network systems are shared resources the designer must provide mechanisms to protect users from each other according to a *policy.*

It is our belief that, as in this list, security is often left until last in the design process, which results in not enough attention and emphasis being given to security. If security is *designed in*, it can simply be made part of the design space in which we search for attractive cost/performance tradeoffs. For example, if acceptable flexibility requires downloadable software, and acceptable security means that only trusted downloadable software will be loaded, our cost and performance optimizations will reflect ideas such as minimizing dynamic checks

with static pre-checks or other means. If security is not an issue, there is no point in doing this.

The designer's major challenge is finding a point (or set of points) in the design space which is acceptable to a large enough market segment to influence the community of users. Sometimes this is not possible; the commercial emphasis on forwarding performance is so overwhelming that concessions to security slowing the transport plane are simply unacceptable. Fortunately, organizations have become sufficiently dependent on information networks that *security does sell*.

In the context of active networks, the major focus of security is the set of activities which provide flexibility; that is, the facility to inject new code "on-the-fly" into network elements. To build a secure infrastructure, first, the infrastructure itself (the "checker") must be unaltered. Second, the infrastructure must provide assurance that loaded modules (the dynamic checking) will not violate the security properties. In general, this is very hard. Some means currently under investigation include domain-specific languages which are easy to check (*e.g.*, PLAN), proof-carrying code [NL96, Nec97], restricted interfaces (ALIEN), and distributed responsibility (SANE). Currently, the most attractive point in the design space appears to be a restricted domain-specific language coupled to an extension system with heavyweight checks. In this way, the frequent (per-packet) dynamic checks are inexpensive, while focusing expensive scrutiny on the extension process. This idea is manifest in the SwitchWare active network architecture [AAH+98].

2.3 Possible Approaches

Security of Active Networks is a broad evolving area. We will mention only some of the most directly relevant related work. In addition to the related works sections of the papers listed, we suggest Moore [Moo98] as a source of additional information in this area.

Software fault isolation as a safety mechanism for mutually-suspicious modules running in the same address space was introduced in [WLAG93]. This technique involves inserting run-time checks in the application code. While it has been successfully demonstrated for RISC architectures, application of the same techniques to CISC architectures remains problematic.

Typed assembly language [MWCG98] propagates type safety information to the assembly language level, so assembly code can be verified. However, there are several security properties (*e.g.*, resource usage, which is a dynamic measure) that do not easily map into the type-checking model because of the latter's static nature.

Proof-carrying code [Nec97] permits arbitrary code to be executed as long as a valid proof of safety accompanies it. While this is a very promising technique, it is not clear that all desirable security properties and policies are expressible and provable in the logic used to publish the policy and encode the proof. Used in conjunction with other mechanisms, we believe that it will prove a very useful security tool.

PLAN [HKM+98, HM] is a part of the SwitchWare [AAH+98, SFG+96] project at the University of Pennsylvania. The PLAN project is investigating the tradeoffs brought about by using a different language for active packets than is used for active extensions. They have designed a new language called PLAN (which is loosely based on ML [MTH90]). PLAN is designed so that pure PLAN programs will not be able to violate the security policy. This policy is intended to be sufficiently restrictive that node administrators will be willing to allow PLAN programs to run without requiring authentication. Because this limits the operations that can be performed, PLAN programs can call *services* which can either be active extensions or facilities built into the system. These services may require authentication and authorization before allowing access to the resources they protect.

The Safetynet Project [WJGO98] at the University of Sussex has also designed a new language for active networking. They have explicitly enumerated what they feel are the important requirements for an active networking language and then set about designing a language to meet those requirements. In particular, they differ from PLAN in that they hope to use the type system to allow safe accumulation of state. They appear to be trying to avoid having any service layer at all.

Java [GJS96] and ML [MTH90, Ler] (and the MMM [Lou96] project) provide security through language mechanisms. More recent versions of Java provide protection domains [GS98]. Protection domains were first introduced in Multics [Sch72, Sch75, MSS77, Sal74]. These solutions are not applicable to programs written in other languages (as may be the case with a heterogeneous active network with multiple execution environments), and are better suited for the applet model of execution than active networks. The need for a separate bytecode verifier is also considered by some a disadvantage, as it forces expensive (in the case of Java, at least) language-compliance checks prior to execution. In this area, there is some research in enhancing the understanding of the tradeoffs between compilation time/complexity, and bytecode size, verification time, and complexity.

It should be noted that language mechanisms can (and sometimes do) serve as the basis of security of an active network node. Other language-based protection schemes can be found in [BSP+95, CLFL94, HCC98, LOW98, LR99, GB99].

3 SANE Architecture

Previous attempts at system security have not taken a holistic approach. The approaches typically focused on a major component of the system. For instance, operating system research has usually ignored the bootstrap process of the host. As a result, a *trustworthy* operating system is started by an *untrustworthy* bootstrap! This creates serious security problems since most Operating Systems require some lower level services, *e.g.,* firmware, for *trustworthy* initialization and operation. A major design goal of SANE [AAKS98a] was to reduce the number and size of components that are assumed as *trustworthy*. A second major design

goal of SANE was to provide a secure and reliable mechanism for establishing a security context for active networking. An application or node could then use that context in any manner it desired.

No practical system can avoid assumptions, however, and SANE is no different. Two assumptions are made by SANE. The first assumption is that the physical security of the host is maintained through strict enforcement of a physical security policy. The second assumption SANE makes is the existence of a Public Key Infrastructure (PKI). While a PKI is required, no assumptions are made as to the type of PKI, *e.g.,* hierarchical or "web of trust."[Com89, LR97, Zim95, BFIK98, BFIK99]

The overall architecture of SANE for a three-node network is shown in Figure 2.

The initialization of each node begins with the bootstrap. Following the sucessful completion of the bootstrap, the operating system is started which loads a general purpose evaluator, *e.g.,* a Caml [Ler] or Java [GJS96] runtime. The evaluator then starts an "Active Loader" which restricts the environment provided by the evaluator. Finally, the loader loads an "Active Network Evaluator" (ANE) which accepts and evaluates active packets, *e.g.,* PLAN [HKM+98], Switchlet, or ANTS [WGT98]. The ANE then loads the SANE module to establish a security context with each network neighbor. Following the establishment of the security context, the node is ready for secure operation within the active network.

It should be noted that the services offered by SANE can be used by most active networking schemes. In our current system, SANE is used in conjunction with the ALIEN architecture [Ale98]. ALIEN is built on top of the Caml runtime, and provides a network bytecode loader, a set of libraries, and other facilities necessary for active networking.

The following sections describe the three components of SANE. These include the AEGIS [AFS97, AKFS98] bootstrap system, the ALIEN [Ale98] architecture, and SANE [AAH+98, AAKS98a] itself.

3.1 AEGIS Bootstrap

AEGIS [AFS97] modifies the standard IBM PC process so that all executable code, except for a very small section of trustworthy code, is verified prior to execution by using a digital signature. This is accomplished through modifications and additions to the BIOS (Basic Input/Output System). In essence, the trustworthy software serves as the root of an authentication chain that extends to the evaluator and potentially beyond, to "active" packets. In the AEGIS boot process, either the Active Network element is started, or a recovery process is entered to repair any integrity failure detected. Once the repair is completed, the system is restarted to ensure that the system boots. This entire process occurs without user intervention. AEGIS can also be used to maintain the hardware and software configuration of a machine.

It should be noted that AEGIS does not verify the correctness of a software component. Such a component could contain an exploitable flaw. The goal of

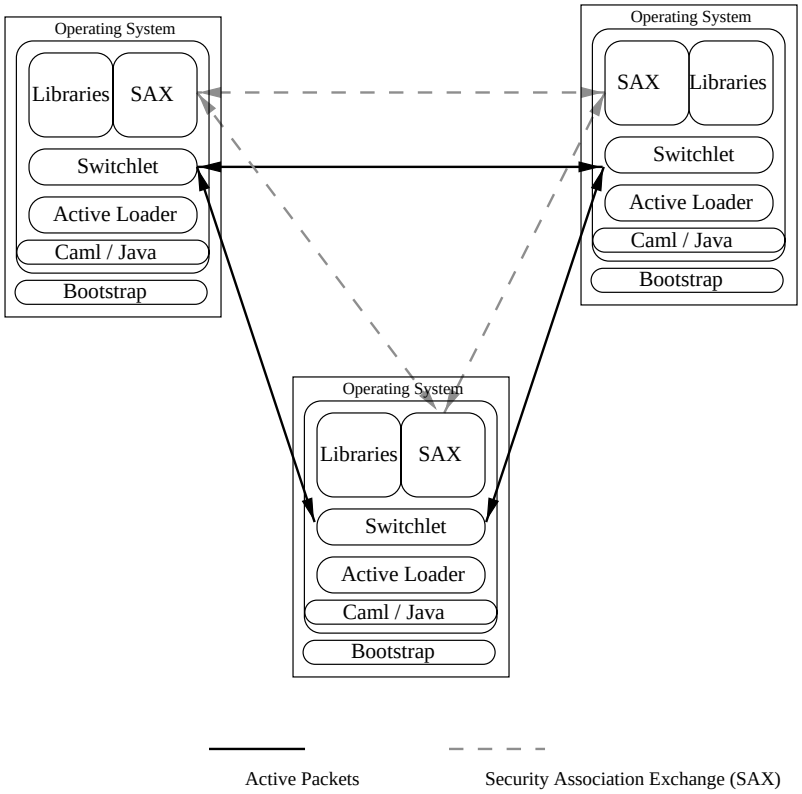

Fig. 2. SANE Network Architecture

AEGIS is to prevent tampering of components that are considered trustworthy by the system administrator. AEGIS verifies the integrity of already trusted components. The nature of this trust is outside the scope of this paper.

Other work on the subject of secure bootstrapping includes [TY91, Yee94, Cla94, LAB92, HKK93]. A more extensive review of AEGIS and its differences with the above systems can be found in [AFS97, AKFS98].

AEGIS Layered Boot and Recovery Process. AEGIS divides the boot process into several levels to simplify and organize the BIOS modifications, as shown in Figure 3. Each increasing level adds functionality to the system, providing correspondingly higher levels of abstraction. The lowest level is Level 0. Level 0 contains the small section of trustworthy software, digital signatures, public key certificates, and recovery code. The integrity of this level is assumed as valid. We do, however, perform an initial checksum test to identify PROM failures. The first level contains the remainder of the usual BIOS code and the

CMOS. The second level contains all of the expansion cards and their associated ROMs, if any. The third level contains the operating system boot sector. These are resident on the bootable device and are responsible for loading the operating system kernel. The fourth level contains the operating system, and the fifth and final level contains the ALIEN architecture and other active nodes..

The transition between levels in a traditional boot process is accomplished with a jump or a call instruction without any attempt at verifying the integrity of the next level. AEGIS, on the other hand, uses public key cryptography and cryptographic hashes to protect the transition from each lower level to the next higher one, and its recovery process through a trusted repository ensures the integrity of the next level in the event of failures [AKFS98].

The trusted repository can either be an expansion ROM board that contains verified copies of the required software, or it can be another Active node. If the repository is a ROM board, then simple memory copies can repair or shadow failures. In the case of a network host, the detection of an integrity failure causes the system to boot into a recovery kernel contained on the network card ROM. The recovery kernel contacts a "trusted" host through the secure protocol described in [AKFS98, AKS98] to recover a signed copy of the failed component. The failed component is then shadowed or repaired, and the system is restarted (warm boot).

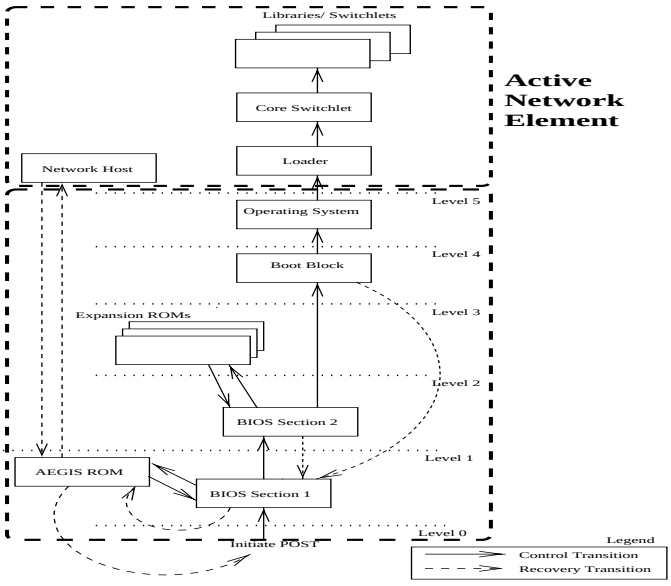

Fig. 3. AEGIS boot control flow

3.2 The ALIEN Architecture

The basis of the ALIEN approach is that we take a general model of computation and restrict it. Caml provides the general model and ALIEN provides the restrictions.

More precisely, Caml provides a model of computation equivalent to that of a Turing machine. By itself, this computation model is secure since it involves no shared resources. In practice, since we are running on a real machine, we have denial-of-service attacks that arise because our CPU and memory resources are finite. Additionally, the actual Caml environment also includes a runtime system that, among other features, provides access to operating system primitives, which, in turn, provide access to shared resources. Further, under this runtime, memory is a shared resource. The role of ALIEN is to control the access to these shared resources and thereby ensure that a loaded program (called *"switchlet"*) does not exceed its resource limits (ALIEN is not responsible for determining those limits).

ALIEN itself is built of three major components. The Loader provides the interface to the Objective Caml runtime system. The Core Switchlet builds on the Loader both by providing the security-related restrictions required and by providing more generally useful interfaces to low-level functions. Finally, the libraries are sets of utility routines. Each of these pieces will be briefly covered in turn in the following paragraphs.

The Loader. The Loader provides the core of ALIEN's functionality. It provides the interface to the operating system (through the language runtime) plus some essential functions to allow system startup and loading of switchlets, as shown in Table 1. Thus, it defines the "view of the world" for the rest of ALIEN. Moreover, since security involves interaction with either the external system or with other switchlets, the Loader provides the basis of security.

It should be noted that Loader provides mechanisms rather than policy; policies in the Core Switchlet can be changed by changing pieces of the Core Switchlet.

startup routines	initialize system
switchlet loading	dynamically load switchlets consistent with ALIEN security
system console	console read loop

Table 1. Loader functionality

The Core Switchlet. Above the Loader is the Core Switchlet. It is responsible for providing the interface that switchlets see. It relies upon the Loader for access to operating system resources, and then layers additional mechanisms to add security and, often, utility. In providing an interface to switchlets, it

determines the security policies of the system. By including or excluding any function, it can determine what switchlets can or cannot do. Since it is loadable, the administrator can change or upgrade its pieces as necessary. This also allows for changes in the security policy.

The policies of the Core Switchlet are enforced through a combination of *module thinning* and *type safety*. Type safety ensures that a switchlet can only access data or call functions that it can name. This allows implementations of ALIEN that run in a single address space, thus avoiding the overheads normally associated with crossing hardware-enforced boundaries. [NFP99].

Module thinning allows the Core Switchlet to present a limited interface to switchlets. Combining this with type safety, switchlets can be prevented from calling functions or accessing data even though they share an address space. It is even possible to differentiate switchlets so as to provide a rich interface to a trusted switchlet or to provide a very limited interface to an anonymous switchlet. Similar approaches have been taken in [LR99, BSP+95, vE99].

In many ways, the interface that the Core Switchlet presents to switchlets and libraries is like the system call interface that a kernel presents to applications. Through design of the interface the system can control access to underlying resources. With a well-designed interface, the caller can combine the functions provided to get useful work done. Table 2 shows the functionality provided by the Core Switchlet.

language primitives	policy for access to the basic functions of the language
operating system access	policy for access to the operating system calls
network access	policy and mechanism for access to the network
thread access	policy for access to threads primitives
loading support	policy and mechanism to support loading of switchlets
message logging	policy and mechanism for adding messages to the log file

Table 2. Core Switchlet functionality

Because we are implementing a network node, access to the network is particularly important. Generally this consists of allowing switchlets to discover information about the interfaces on the machines and the attached networks, receive packets, and send packets. One element of this task which is particularly important is the demultiplexing of incoming packets. The Core Switchlet must be able to determine whether zero, one, or more than one switchlet is interested in an arriving packet. If more than one switchlet is interested in the packet, policy should dictate which switchlet or switchlets receive a copy of the packet. Security is an important element of the decision as a switchlet should be able to be certain that it will get all packets that it should receive under the policy, and should not be able to get any packets that it should not receive under the policy. Without such security, denial-of-service attacks and information stealing are quite easy.

The Library. The library is a set of functions which provide useful routines that do not require privilege to run. The proper set of functions for the library is a continuing area of research. Some of the things that are in the library for the experiments we have performed include utility functions and implementations of IP and UDP [Pos80].

Locating Functionality. When expanding our implementation, it is not always obvious in which layer the new functionality belongs. In this section, we present the principles we use to make this determination. Our first principle is that if the functionality can be implemented in a library, it should be. Said another way, if the functions exposed by the Core Switchlet or available from other libraries provide the infrastructure needed to implement the new functionality, a library is warranted.

If the new functionality relies on some element of the runtime not made available to unprivileged code, then either the Loader or the Core Switchlet must be expanded. Because these elements define the common, expected interface available at the switch, we attempt to keep them small to minimize the required resources. Therefore, our second principle is that we prefer to break off the smallest reasonable portion of the new functionality (consistent with security) that can be implemented in the privileged parts of the system. The remainder becomes a library. In our experience this also aids generality, as the privileged portion is often useful to other libraries developed later. For example, to implement IP, we built a small module inside the Core Switchlet which reads Ethernet frames from the operating system. It also demultiplexes the frames based on the Ethernet type field to increase generality. The remainder of IP, which processes headers, could then be made a non-privileged library.

Our third principle is that if this privileged functionality sets policy, it needs to go into the Core Switchlet. As discussed above, policy must be set in the Core Switchlet so that the loading mechanism can be used as needed to change policy. Our final principal is any functionality that provides pure mechanism is placed in the Core Switchlet unless it is needed before the Core Switchlet can be running.

3.3 SANE Services

SANE builds on AEGIS and ALIEN in order to provide security services for an active network. We believe that these services are required for the deployment of a robust active infrastructure. This is not to say that they contain all the security mechanisms one would ever want. Rather, they are basic building blocks needed for possibly more advanced mechanisms. These services include:

- Cryptographic primitives, provided as ALIEN libraries. A number of symmetric [NBS77] and public-key [NIS94] cryptosystems and hash [NIS95] functions are made available for use by programmers and other system components. These are building blocks for cryptographic protocols and other mechanisms that provide higher-order security primitives.

- Packet authentication. This can be achieved through signing the packet with a public-key algorithm, or using some secret key method, like a MAC. Authentication can be used in a number of different contexts:
  - When dynamically loading switchlets over the network, to ensure code integrity and proper access authorization/resource allocation.
  - Similarly, when transmitting data over the network, to ensure data integrity, and packet-flow isolation. This latter may form the basis for economic traffic-management schemes, if the authentication mechanisms prove lightweight enough.

  Public-key authentication allows for zero-roundtrip authentication, since no negotiation is required, but is relatively heavy-weight computationally and is subject to some replay attacks in the absence of node-persistent state or synchronized clocks [Syv94, Gon92] in the network switches. This form of authentication is well-suited to mobile-agent types of applications, such as some active network management schemes [PJ96], or where data generated on a switch needs to be securely combined with the code.
  Secret-key based authentication is faster, and thus better suited for bulk transport. For scalability reasons however, it needs to be automated, and thus needs some public-key infrastructure to base the authentication on. The cost of the automated key establishment can be easily amortized after transmitting even a small amount of data.
- Packet confidentiality (encryption). The same issues with regards to public *vs.* secret key authentication are present here. The uses of this service are analogous to the packet authentication service.
- A key establishment protocol (KEP), which allows two principals in the network to establish secret keys and exchange certificates. The protocol is also used in bootstrap failure-recovery in AEGIS [AKFS98] and is based on the Station-to-Station [DvOW92] protocol, using Diffie-Hellman [DH76] key exchange and DSA [NIS94] (or other) public-key signatures. This protocol is used in three different roles:
  1. Secure boostrap component recovery in AEGIS, as we discussed in Section3.1.
  2. Secure neighbor discovery once the node boots. Similar to verifying the network cards and software components in AEGIS, at this stage the node verifies the immediate network topology and establishes trust between "adjacent" switches. The shared secret keys established in this manner can be used for hop-by-hop secure transmission of data or code, which is a requirement for some mobile-agent types of applications. Other critical infrastructure information can be secured by this mechanism as well, *e.g.,* routing updates.
  3. Session-key establishment, principal authentication and authorization. The shared secret keys can be used to secure the communications of any two principals in the network. Furthermore, principals can authenticate each other and exchange authorization credentials. For example, a user can verify the identity of a switch he needs to load a program on, while the switch can determine whether the user has permission to do so, and what other restrictions apply. We make use of

KeyNote [BFIK98, BFIK99] credentials to provide this functionality. These credentials specify the resource usage and access control policies that ALIEN enforces.

- Administrative domains allow a set of network elements under the same administrative control to restrict security requirements when communicating with each other. In such a configuration, "border" elements act as present-day firewalls and can require *e.g.,* a number of iterations of KEP to establish all the problem credentials. They can then mark the active packets such that further negotiation in order to determine credentials is not needed by the "interior" switches. In essense, these "active firewalls" act as introducers of "outsiders" in a closed system. Administrative domains are built on top of secure packet exchange, in conjunction with the key management protocol.
- Naming services allow for unsupervised but collision-free[1] (secure) identification of programs. The basis of this approach is to combine hashes of the code, public keys (and signatures), and user-defined strings to generate "names" for pieces of code. Thus, a certain program can have different names, each with different semantics and trust dependencies. Such a service is necessary in an active network environment where different users' modules can explicitly interact with, or even depend on, each other. For more details, see [AAKS98a].

4 Conclusions and Future Work

This paper has presented an overview of the programmable network architectures called "Active Networks" and illustrated the security issues facing the architects of such systems. While the goal of Active Networks is to increase the set of design options available in distributed systems, the flexibility of a programmable infrastructure introduces considerable burdens for control and management. We have outlined, and illustrated in some detail, strategies for coping with some of these issues. Many issues currently remain unresolved, and are thus important directions for future work.

First and foremost is the requirement that newly configured functionality not damage the network as an aggregate, in addition to not damaging the network element into which it is configured. The means for attacking such problems remains unclear, but two of the more promising approaches are the use of formal languages and methods for distributed concurrent systems (such as CSP [Hoa78, Hoa84]), and the use of economic methods to deal with aggregate costs which accrue in a distributed fashion.

The second important direction is time-sensitive resource access, where Quality of Service (QoS) is part of the resource access model to which the security architecture applies. Much of the fundamental work of computer security has been based on a time-independent model of resource access, to which symbolic

[1] To the extent that the cryptographic hash functions employed are resistant to collisions.

logic and formal methods can be applied. However, as any user of information, whether "secure" or "insecure" is well aware, if the information doesn't get there in time, it is useless. If the service is not provided, it has failed. Thus we have to push our security models to reflect time, so that service (or the attackers goal, "denial-of-service") is as first class as identification or access control in network security architectures.

Finally, we must worry about scale. Some of the solutions we have discussed here work well in the small but must be automated to permit scaling to large systems. For example, the SANE model of permitting resource access to trusted entities presumes trust establishment and trust specification. In a (multi-) million node network, such trust management must be carried out automatically, and must be globally specified using human-comprehensible policies rather than node-node relationships.

5 Acknowledgements

Portions of this paper are updated from [AAKS98a] and [AAKS98b]. This work was supported by DARPA under Contract #N66001-96-C-852, with additional support from the Intel Corporation.

References

[AAH+98] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):29–36, 1998.

[AAKS98a] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):37–45, 1998.

[AAKS98b] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. Safety and Security of Programmable Network Infrastructures. *IEEE Communications Magazine*, 36(10):84 – 92, 1998.

[AFS97] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[AKFS98] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated Recovery in a Secure Bootstrap Process. In *Proceedings of Network and Distributed System Security Symposium*, pages 155–167. Internet Society, March 1998.

[AKS98] W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. DHCP++: Applying an efficient implementation method for fail-stop cryptographic protocols. In *Proceedings of Global Internet (GlobeCom) '98*, November 1998.

[Ale98] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, September 1998.

[Atk95] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[BFIK98] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System. Work in Progress, http://www.cis.upenn.edu/~angelos/keynote.html, June 1998.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming* [VJ99], pages ??–??

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the spin operating system. In *Proc. 15th SOSP*, pages 267–284, December 1995.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Internet RFC 2208, 1997.

[Cla94] Paul Christopher Clark. *BITS: A Smartcard Protected Operating System.* PhD thesis, George Washington University, 1994.

[CLFL94] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. In *ACM Transactions on Computer systems*, November 1994.

[Com89] Consultation Committee. *X.509: The Directory Authentication Framework.* International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[Dee89] S. E. Deering. Host extensions for IP multicasting. Internet RFC 1112, 1989.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, Nov 1976.

[DOD85] DOD. Trusted Computer System Evaluation Criteria. Technical Report DOD 5200.28-STD, Department of Defense, December 1985.

[DRI96] Daemon9, Route, and Infinity. Project neptune. *Phrack Magazine*, 7(48), 1996.

[DvOW92] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[EFR+97] Carl M. Ellison, Bill Frantz, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple Public Key Certificate. Work in Progress, http://www.pobox.com/~cme/html/spki.html, April 1997.

[GB99] R. Grimm and B. Bershad. Providing policy neutral and transparent access control in extensible systems. In *Secure Internet Programming* [VJ99], pages ??–??

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, Reading, 1996.

[Gon92] L. Gong. A Security Risk of Depending on Synchronized Clocks. *ACM Operating Systems Review*, 26(1), January 1992.

[GS98] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 125–134, March 1998.

[HCC98] C. Hawblitzel, C. Chang, and G. Czajkowski. Implementing Multiple Protection Domains in Java. In *Proc. of the 1998 USENIX Annual Technical Conference*, pages 259–270, June 1998.

[HKK93] Hermann Härtig, Oliver Kowalski, and Winfried Kühnhauser. The Birlix security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[HM] Mike W. Hicks and Jonathan T. Moore. PLAN Web Page. http://www.cis.upenn.edu/~switchware/PLAN/.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa84] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1984.

[LAB92] Butler Lampson, Martin Abadi, and Michael Burrows. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, v10:265–310, November 1992.

[Ler] Xavier Leroy. The Caml Special Light System (Release 1.10). http://pauillac.inria.fr/ocaml.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[Lou96] François Louaix. A Web Navigator with Applets in Caml. In *Fifth WWW Conference*, 1996.

[LOW98] J. Y. Levy, J. K. Ousterhout, and B. B. Welch. The Safe-Tcl Security Model. In *Proc. of the 1998 USENIX Annual Technical Conference*, pages 271–282, June 1998.

[LR97] B. Lampson and R. Rivest. Cryptography and Information Security Group Research Project: A Simple Distributed Security Infrastructure. Technical report, MIT, 1997.

[LR99] X. Leroy and F. Rouaix. Security properties of typed applets. In *Secure Internet Programming* [VJ99], pages ??–??

[MMO+94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical report, Department of Computer Science, University of Arizona, June 1994.

[Moo98] J. Moore. Mobile Code Security Techniques. Technical Report MS-CIS-98-28, University of Pennsylvania, May 1998.

[MSS77] D.D. Clark M.D. Schroeder and J.H. Saltzer. The MULTICS Kernel Design Project. In *Sixth ACM Symposium on Operating Systems Principles*, pages 43–56, 1977.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[NBS77] Data Encryption Standard, January 1977.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119. ACM Press, New York, January 1997.

[Neu95] Peter G. Neumann. Architectures and Formal Representations for Secure Systems. Final Report. SRI Project 6401 A002, SRI International, October 1995.

[NFP99] R. De Nicola, G. L. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming* [VJ99], pages ??–??

[NIS94] Digital Signature Standard, May 1994.

[NIS95] Secure Hash Standard, April 1995. Also known as: 59 Fed Reg 35317 (1994).

[NL96] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI)*, pages 229–243. Usenix, Seattle, 1996.

[Pan96] Cracker Attack Paralyzes PANIX. RISKS Digest. Volume 18. Issue 45., September 1996.

[PJ96] C. Partridge and A. Jackson. Smart Packets. Technical report, BBN, 1996. http://www.net-tech.bbn.com-/smtpkts/smtpkts-index.html.

[Pos80] Jon Postel. User Datagram Protocol. Internet RFC 768, 1980.

[Pos81] Jon Postel. Internet Protocol. Internet RFC 791, 1981.

[Sal74] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. In *Communications of the ACM*, pages 388–402, July 1974.

[Sch72] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility.* PhD thesis, MIT, September 1972.

[Sch75] M.D. Schroeder. Engineering a Security Kernel for MULTICS. In *Fifth Symposium on Operating Systems Principles*, pages 125–132, November 1975.

[SFG+96] J. M. Smith, D. J. Farber, C. A. Gunter, S. M Nettles, D. C. Feldmeier, and W. D. Sincoskie. SwitchWare: Accelerating Network Evolution. Technical Report MS-CIS-96-38, CIS Dept. University of Pennsylvania, 1996.

[Syv94] P. Syverson. A Taxonomy of Replay Attacks. In *Proceedings of the Computer Security Foundations Workshop VII (CSFW7)*, June 1994.

[TY91] J.D. Tygar and Bennet Yee. DYAD: A System for Using Physically Secure Coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[vE99] T. von Eicken. J-kernel a capability based operating system for java. In *Secure Internet Programming* [VJ99], pages ??–??

[VJ99] Jan Vitek and Christian Jensen. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects.* Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 1999.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *IEEE OpenArch Proceedings.* IEEE Computer Society Press, Los Alamitos, April 1998.

[WJGO98] Ian Wakeman, Alan Jeffrey, Rory Graves, and Tim Owen. Designing a Programming Language for Active Networks. *submitted to Hipparch special issue of Network and ISDN Systems*, June 1998. http://www.cogs.susx.ac.uk/projects/safetynet/papers/isdn.ps.gz.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proc. of the 14th Symposium on Operating System Principles*, pages 203–216, December 1993.

[Yee94] Bennet Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.

[Zim95] P. Zimmerman. PGP User's Manual, 1995.